Journal of Discrete Algorithms 16 (2012) 151–169

Contents lists available at SciVerse ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

p-Suffix sorting as arithmetic coding ☆

Richard Beal *, Donald Adjeroh

West Virginia University, Lane Department of Computer Science and Electrical Engineering, Morgantown, WV 26506, United States

ARTICLE INFO

Article history:
Available online 23 May 2012

Keywords:
Parameterized suffix array
Parameterized suffix sorting
Arithmetic coding
Fingerprints
p-string
p-match

ABSTRACT

The challenge of direct parameterized suffix sorting (p-suffix sorting) for a parameterized string (p-string), say T of length-n, is the dynamic nature of the n parameterized suffixes (p-suffixes) of T. In this work, we propose transformative approaches to direct p-suffix sorting by generating and sorting lexicographically numeric fingerprints and arithmetic codes that correspond to individual p-suffixes. Our algorithm to p-suffix sort via fingerprints is the first theoretical linear time algorithm for p-suffix sorting for non-binary parameter alphabets, which assumes that, in practice, all codes are within the range of an integral data type. We eliminate the key problems of fingerprints by introducing an algorithm that exploits the ordering of arithmetic codes to sort p-suffixes in linear time on average. The arithmetic coding approach is further extended to handle p-strings in the worst case. This algorithm is the first direct p-suffix sorting approach in theory to execute in $o(n^2)$ time in the worst case, which improves on the best known theoretical result on this problem that sorts p-suffixes based on p-suffix classifications in $O(n^2)$ time. We show that, based on the algorithmic parameters and the input data, our algorithm does indeed execute in linear time in various cases, which is confirmed with experimental results.

1. Introduction

Conventional pattern matching involves the matching of standard strings over an alphabet Σ. Parameterized pattern matching using parameterized strings, introduced by Baker [4], attempts to answer pattern matching questions beyond its classical counterpart. A parameterized string (p-string) is a production of symbols from the alphabets Σ and Π, which represent the constant symbols and parameter symbols respectively. Given a pair of p-strings S and T, the parameterized pattern matching (p-match) problem is to verify whether the individual constant symbols match and whether there exists a bijection between the parameter symbols of S and T. If the two conditions are met, S is said to be a p-match of T. For example, there exists a p-match between the p-strings $z = y * f / + + y$; and $a = b * f / + + b$; that represent program statements over the alphabets $\Sigma = \{*, /, +, =, ;\}$ and $\Pi = \{a, b, f, y, z\}$. Applications inherent to the p-matching problem include detecting plagiarism in academia and industry, reporting similarities in biological sequences [27], discovering cloned code segments in a program to assist with software maintenance [4], and answering critical legal questions regarding the unauthorized use of intellectual property [29].

Initial solutions to the p-match problem were based on the parameterized suffix tree (p-suffix tree) [4]. Idury et al. [17] studied the multiple p-match problem using automata. The physical space requirements of the p-suffix tree led to algorithms such as parameterized-KMP [3], parameterized-BM [6], and the parameterized suffix array (p-suffix array) [16,12]. Analogous

☆ A shorter version of this paper was presented at the International Workshop on Combinatorial Algorithms, Victoria, BC, 2011 (Beal and Adjeroh, 2011 [9]). This work was partly supported by grants from the National Historical Publications & Records Commission, and from WV-EPSCoR RCG.

* Corresponding author.
E-mail addresses: r.beal@computer.org (R. Beal), don@csee.wvu.edu (D. Adjeroh).

1570-8667/$ – see front matter
http://dx.doi.org/10.1016/j.jda.2012.05.001

to standard suffix sorting, the problem of parameterized suffix sorting (p-suffix sorting) is to sort all the n parameterized suffixes (p-suffixes) of an n-length p-string into a lexicographic order. The major difficulty is that unlike traditional suffixes of a string, the p-suffixes are dynamic, varying with the starting position of the p-suffix. Thus, standard suffix sorting approaches cannot be directly applied to the p-suffix sorting problem. Current approaches to directly construct the p-suffix array *without* a p-suffix tree for an n-length p-string from an arbitrary alphabet require $O(n^2)$ and $O(n^3)$ time in the worst case [16]. In [16], a standard quicksort was used to sort p-suffixes naïvely in $O(n^3)$ time. An improved algorithm was given in [16] to sort p-suffixes via classification and bucket or radix sorting in $O(n^2)$ time. Prior to our work, the group of I, Deguchi et al. [16,12] was the only known group with results on the direct p-suffix sorting problem. The existence of a better theoretical p-suffix sorting algorithm is proposed as an open problem in [16].

The *direct* p-suffix sorting problem is significant because of two major reasons. First, the p-suffix array, like the traditional suffix array, is a lightweight data structure in terms of space. If we traverse the p-suffix tree to *indirectly* obtain the p-suffix array, we must allocate the heavyweight memory footprint required for the tree, voiding any space advantage of the array representation. In a case where the tree is readily accessible, it is possible to obtain the n indices at the cost of a factor of the alphabet in either the time or space depending on the representation of outgoing edges, i.e. vectors, linked lists, or balanced binary trees. Second, traditional suffix sorting approaches benefit from successive doubling used in [24] or induction [1,2,18,25]. These techniques work with traditional strings because a suffix of a regular string shares information common to other suffixes in the string. This is not necessarily the case with p-suffixes because of their dynamic nature or more formally, the fact that the p-suffix at some position, say k of the p-string T, is not necessarily a suffix or even a substring of the p-suffix at position $(k-1)$, $(k-2)$, or even 1. These obstacles introduce many challenges to the p-suffix sorting problem. In this paper, we improve the running time of p-suffix sorting and address the open problem proposed in [16].

Main Contribution. We construct p-suffix arrays by generating and sorting m-length codes that represent the individual p-suffixes of a p-string. We propose the first theoretical linear time claims to directly p-suffix sort p-strings on average from non-binary parameter alphabets. Further, we propose the first direct p-suffix sorting solution to execute in $o(n^2)$ time in the worst case. We state our main result in the following, where m is the size of the block used in sorting and h is a measure of the number of m-blocks needed to clearly differentiate the p-suffixes:

Theorem 21. *Given a p-string T of length n, p-suffix-sorting of T can be accomplished in $O(n)$ time and $O(n)$ space on average via parameterized arithmetic coding.*

Theorem 26. *Given a p-string T of length n and an integer m with $1 \leqslant m \leqslant n$, the p-suffix-sorting of T is accomplished in $O(n)$ time on average and precisely $O(\sum_{i=1}^{h-1}[(n-i\times m)\log(n-i\times m)]+n\log n)$ time in the worst case via parameterized arithmetic coding. Worst case space is $O(n)$.*

Corollary 27. *Let $\epsilon > 0$ be a very small number $\epsilon = 0.00\ldots1$. For a p-string T of length n and a chosen $m = O(\log^{1+\epsilon} n)$, then in the worst case when $h = O(\frac{n}{m})$ the running time of Algorithm 6 is $o(n^2)$.*

2. Background/related work

Baker [4] defines pattern matching as either: (1) exact matching, (2) parameterized-matching, or (3) matching with modifications. A parameterized match (p-match) is a sophisticated matching scheme based on the composition of a parameterized string (p-string). A p-string is composed of symbols from a constant symbol alphabet Σ and a parameter alphabet Π. A pair of p-strings S and T of length n are said to p-match when the constant symbols $\sigma \in \Sigma$ match and there exists a bijection of parameter symbols $\pi \in \Pi$ between the pair of p-strings. Baker [4] offered the first p-match breakthroughs, namely, the `prev` encoding to detect a p-match and the parameterized suffix tree (p-suffix tree) analogous to the suffix tree for traditional strings [1,15,28]. The p-suffix tree is built on the `prev` encodings of the suffixes of the p-string, demanding $O(n(|\Pi| + \log(|\Pi| + |\Sigma|)))$ construction time in the worst case [5]. Improvements to the p-suffix tree construction were introduced by Kosaraju [21]. Other contributions in the area of parameterized suffix trees include construction via randomized algorithms [10,22,23]. Like the traditional suffix tree [1,15,28], the p-suffix tree [4] implementation suffers from a large memory footprint. Other solutions that address the p-match problem without the space limitations of the p-suffix tree include the parameterized-KMP [3] and parameterized-BM [6], variants of traditional pattern matching approaches. Such p-matching approaches always match a pattern across the text. As a result, these approaches are best suited for infrequent use: to return the location, maximum length, or existence of a p-match. Alternatively, suffix structures (suffix arrays and suffix trees) return a data structure with information about the suffixes of a string to assist in efficient pattern matching and are best suited for applications with a heavy demand on matching.

The native time and space efficiency of the suffix array led to the origination of the parameterized suffix array (p-suffix array). The p-suffix array is analogous to the suffix array for traditional strings introduced in [24]. Manber and Myers [24] show how to combine the suffix array and the *LCP* (longest common prefix) array to competitively search for pattern $P = P[1\ldots m]$ in a text $T = T[1\ldots n]$ in $O(m+\log n)$ time. Direct p-suffix array construction was first introduced by Deguchi

et al. [12] for binary strings requiring $O(n)$ construction time through the assistance of a defined `fw` encoding. Deguchi and colleagues [16] later proposed the first solutions to *direct* p-suffix sorting with an arbitrary alphabet size requiring $O(n^2)$ and $O(n^3)$ time in the worst case, *without* the assistance of a p-suffix tree. The parameterized longest common prefix (*pLCP*) array, analogous to the traditional *LCP*, was also defined and constructed in [16,12]. We show how to compute the *pLCP* array and other data structures for p-strings in [8,7]. In this work, we propose methods to the direct p-suffix sorting problem without the assistance of the p-suffix tree by using fingerprints and coding methods from information theory.

3. Preliminaries

A string on an alphabet Σ is a production $T = T[1]T[2]\dots T[n]$ from Σ^n with $n = |T|$ the length of T. We will use the following string notations: $T[i]$ refers to the ith symbol of string T, $T[i\dots j]$ refers to the substring $T[i]T[i+1]\dots T[j]$, and $T[i\dots n]$ refers to the ith suffix of T: $T[i]T[i+1]\dots T[n]$. The area of parameterized pattern matching defines the finite alphabets Σ and Π. Alphabet Σ denotes the set of constant symbols while Π represents the set of parameter symbols. We assume the use of indexed alphabets. Alphabets are defined such that $\Sigma \cap \Pi = \emptyset$. Furthermore, we append the terminal symbol $\$ \notin \Sigma \cup \Pi$ to the end of all strings to clearly distinguish between suffixes. For practical purposes, we can assume that $|\Sigma| + |\Pi| \leqslant n$ since, otherwise a single mapping can be used to enforce the condition.

Definition 1 *(Parameterized string (p-string)).* A p-string is a production T of length n from $(\Sigma \cup \Pi)^*\$$.

Consider the alphabet arrangements $\Sigma = \{A, B\}$ and $\Pi = \{w, x, y, z\}$. Example p-strings include $S = AxByABxy\$$, $T = AwBzABwz\$$, and $U = AyByAByy\$$.

Definition 2 *(Parameterized matching (p-match)).* (See [4,12].) A p-match exists between pair of p-strings S and T with $n = |S|$ if and only if $|S| = |T|$ and each $1 \leqslant i \leqslant n$ corresponds to one of the following:

1. $S[i], T[i] \in (\Sigma \cup \{\$\}) \wedge S[i] = T[i]$
2. $S[i], T[i] \in \Pi \wedge ((a) \vee (b))$/* parameter bijection */
   (a) $S[i] \neq S[j], T[i] \neq T[j]$ **for any** $1 \leqslant j < i$
   (b) $S[i] = S[i-q]$ **iff** $T[i] = T[i-q]$ **for any** $1 \leqslant q < i$.

In our example, we have a p-match between the p-strings S and T since every constant/terminal symbol matches and there exists a bijection of parameter symbols between S and T. U does not satisfy the parameter bijection to p-match with S or T. The process of p-matching requires the `prev` encoding.

Definition 3 *(Previous (`prev`) encoding).* (See [4,12].) Given $\mathbb{Z}$ as the set of non-negative integers, the function `prev`: $(\Sigma \cup \Pi)^*\$ \to (\Sigma \cup \mathbb{Z})^*\$$ accepts a p-string T of length n and produces a string Q of length n that (1) encodes constant/terminal symbols with the same symbol and (2) encodes parameters to point to **previous** like-parameters. More formally, Q is constructed of individual $Q[i]$ with $1 \leqslant i \leqslant n$ where

$$Q[i] = \begin{cases} T[i], & \textbf{if } T[i] \in (\Sigma \cup \{\$\}) \\ 0, & \textbf{if } T[i] \in \Pi \wedge T[i] \neq T[j] \textbf{ for any } 1 \leqslant j < i \\ i-k, & \textbf{if } T[i] \in \Pi \wedge k = \max\{j | T[i] = T[j],\ 1 \leqslant j < i\}. \end{cases}$$

For a p-string T of length n, the above $O(n)$ space `prev` encoding demands the worst case construction time $O(n \log(\min\{n, |\Pi|\}))$, which follows from the discussions of Baker [4,6] and Amir et al. [3] on the dependency of alphabet Π in p-match applications. Note that with an indexed alphabet and an auxiliary $O(|\Pi|)$ mapping structure, we can construct `prev` in $O(n)$ time. Using Definition 3, our examples evaluate to $\texttt{prev}(S) = A0B0AB54\$$, $\texttt{prev}(T) = A0B0AB54\$$, $\texttt{prev}(U) = A0B2AB31\$$. The relationship between p-strings and the lexicographical ordering of the `prev` encoding is fundamental to the p-match problem.

Definition 4 *(`prev` Lexicographical ordering).* Given the p-strings S and T and two symbols s and t from the encodings $\texttt{prev}(S)$ and $\texttt{prev}(T)$ respectively, the relationships $=$, $\neq$, $<$, and $>$ refer to lexicographical ordering between s and t. We define the ordering of symbols from a `prev` encoding of the production $(\Sigma \cup \mathbb{Z})^*\$$ to be $\$ < \zeta \in \mathbb{Z} < \sigma \in \Sigma$, where each ζ and σ are lexicographically sorted in their respective alphabets. The relationships $=$, $\neq$, $\prec$, and $\succ$ refer to the lexicographical ordering between strings. In the case of $\texttt{prev}(S)$ and $\texttt{prev}(T)$, $\texttt{prev}(S) \prec \texttt{prev}(T)$ when $\texttt{prev}(S)[1] = \texttt{prev}(T)[1], \texttt{prev}(S)[2] = \texttt{prev}(T)[2], \dots, \texttt{prev}(S)[j-1] = \texttt{prev}(T)[j-1], \texttt{prev}(S)[j] < \texttt{prev}(T)[j]$ for some $j, j \geqslant 1$. Similarly, we can define $=_k$, $\neq_k$, $\prec_k$, and $\succ_k$ to refer to the lexicographical relationships between a pair of p-strings considering only the first $k \geqslant 0$ symbols.

The following proposition essential to the p-matching problem is directly related to the symbol ordering established in Definition 4.

Proposition 5. *(See [4].) Two p-strings S and T p-match when* $\texttt{prev}(S) = \texttt{prev}(T)$. *Also,* $S \prec T$ *when* $\texttt{prev}(S) \prec \texttt{prev}(T)$ *and* $S \succ T$ *when* $\texttt{prev}(S) \succ \texttt{prev}(T)$.

The example prev encodings show a p-match between S and T since $\texttt{prev}(S) = A0B0AB54\$$ and $\texttt{prev}(T) = A0B0AB54\$$. Also, $U \succ S$ and $U \succ T$ since $\texttt{prev}(U) = A0B2AB31\$ > \texttt{prev}(S) = \texttt{prev}(T) = A0B0AB54\$$. We use the ordering established in Definition 4 to define the parameterized suffix array.

Definition 6 *(Parameterized suffix array (p-suffix array)).* The p-suffix array (pSA) for a p-string T of length n preserves the lexicographical ordering of the indices i representing individual p-suffixes $\texttt{prev}(T[i \ldots n])$ with $1 \leqslant i \leqslant n$, such that $\texttt{prev}(T[pSA[q] \ldots n]) \prec \texttt{prev}(T[pSA[q+1] \ldots n]) \forall q$, $1 \leqslant q < n$. The act of constructing pSA is referred to as p-suffix sorting.

The pSA is analogous to the suffix array SA defined for traditional strings. Let the rank array R rank each p-suffix index in the p-string T to its position in the corresponding pSA or SA. The following $pLCP$ array is used with the pSA for efficient p-matching [16,12,7].

Definition 7 *(Parameterized longest common prefix (pLCP) array).* The $pLCP$ array for a p-string T of length n preserves the length of the longest common prefix between neighboring p-suffixes. We define $\texttt{plcp}(\alpha, \beta) = \max\{k \mid \texttt{prev}(\alpha) =_k \texttt{prev}(\beta)\}$. Then, $pLCP[1] = 0$ and $pLCP[i] = \texttt{plcp}(T[pSA[i] \ldots n], T[pSA[i-1] \ldots n])$, $2 \leqslant i \leqslant n$. Denote $pLCP_{max}$ as the maximum value in the array and denote $pLCP_{\mu}$ as the mean of the array.

The standard LCP and longest common prefix computation lcp are analogous to the $pLCP$ and plcp only without the prev encoding. Also denote LCP_{max} and LCP_{μ} similarly for the standard LCP. For the example $T = AwBzABwz\$$ with $\texttt{prev}(T) = A0B0AB54\$$, we have $pSA = \{9, 8, 7, 4, 2, 1, 5, 6, 3\}$, $pLCP = \{0, 0, 1, 1, 1, 0, 1, 0, 2\}$, $pLCP_{max} = 2$, and $pLCP_{\mu} = 2/3$. The encoding prev is supplemented by the encoding forw.

Definition 8 *(Forward (forw) encoding).* Let the function $\texttt{rev}(T)$ reverse the p-string T and $\texttt{repl}(T, x, y)$ replace all occurrences in T of the symbol x with y. We define the function forw for the p-string T of length n as $\texttt{forw}(T) = \texttt{rev}(\texttt{repl}(\texttt{prev}(\texttt{rev}(T)), 0, n))$.

Essentially, forw performs the following on a p-string T of length n: (1) encodes constant/terminal symbols with the same symbol and (2) encodes each parameter p with the **forward** distance to the next occurrence of p or an unreachable forward distance n. Our definition of the forw encoding generates output mirroring the fw encoding used by Deguchi et al. [16,12]. Let $\mathbb{N}$ refer to the set of positive, non-zero integers. The function $\texttt{fw} : (\Sigma \cup \Pi)^* \to (\Sigma \cup \mathbb{N})^*$ produces an output encoding G with $\texttt{fw}(T) = G$ for each $1 \leqslant i \leqslant n$:

$$G[i] = \begin{cases} T[i], & \textbf{if } T[i] \in \Sigma \\ \infty, & \textbf{if } T[i] \in \Pi \wedge T[i] \neq T[j] \textbf{ for any } i < j \leqslant n \\ k - i, & \textbf{if } T[i] \in \Pi \wedge k = \min\{j \mid T[i] = T[j],\ i < j \leqslant n\} \end{cases}$$

The forw encodings in our example with $n = 9$ are $\texttt{forw}(S) = A5B4AB99\$$, $\texttt{forw}(T) = A5B4AB99\$$, $\texttt{forw}(U) = A2B3AB19\$$.

4. Direct p-suffix sorting motivation

Suffix arrays and suffix trees are data structures that preserve the ordering of the sorted suffixes of a string. Both data structures can be used for similar purposes in pattern matching applications. However, these suffix structures differ by representation and construction. Suffix trees (ST) [1,15,28] represent sorted suffixes in a tree form. With suffix trees, a key construction result is the large practical footprint required by the data structure. Suffix arrays (SA) [24,25] are intended to require less space by preserving the sorted suffixes simply using an integer array. It is possible to construct one suffix structure from the other. Constructing the SA from the ST will eliminate the space saved by the suffix array. In this work, we focus on direct suffix sorting, which adds a restriction to suffix array construction that a suffix tree cannot be used. Traditional suffix array and suffix tree constructions are linear with respect to the text.

In terms of the parameterized string (p-string) [4], direct parameterized suffix sorting (p-suffix sorting) is the construction of the parameterized suffix array (pSA) without the use of the parameterized suffix tree (pST). Currently, several pST constructions [4,5,21,10,22,23] already execute roughly in time linear with respect to the length of the p-string. Current worst case theoretical direct pSA constructions for any alphabet sets require either a naïve cubic or an improved time quadratic to the length of the p-string [12,16]. It is posed as an open question in [16] whether sub-quadratic theoretical worst case algorithms exist for direct pSA constructions.

The construction of the pSA is very different from the construction of the SA because of the fact that the suffixes are drastically different. Recall from the preliminaries that a parameterized suffix (p-suffix) is a suffix under the prev encoding.

Table 1
Suffixes with $\Sigma = \{A, B\}$, $\Pi = \{x, y, z\}$, $n = 9$, $S = ABBABBBA\$$, and $T = yyAxxzyz\$$.

i	Suffix: $S[i \dots n]$	p-suffix: $\texttt{prev}(T[i \dots n])$
1	$ABBABBBA\$$	$01A01052\$$
2	$BBABBBA\$$	$0A01052\$$
3	$BABBBA\$$	$A01002\$$
4	$ABBBA\$$	$01002\$$
5	$BBBA\$$	$0002\$$
6	$BBA\$$	$002\$$
7	$BA\$$	$00\$$
8	$A\$$	$0\$$
9	$\$$	$\$$

A brief analysis of Table 1 shows the core difference between standard suffixes and p-suffixes. That is, standard suffixes are related by simply removing the first symbol to obtain the next suffix. Also, each standard suffix is related to one another. For example, the suffix at 3 is a suffix of 1 and 2. These relationships are exploited in standard direct suffix sorting. From the example in Table 1, we see that this is not the case with p-suffixes and so, standard *SA* constructions do not readily apply. The following lemma formalizes this intricacy with p-suffixes.

Lemma 9. *Given a p-string T of length n, the suffixes of $\texttt{prev}(T)$ are not necessarily the p-suffixes of T. More formally, if $\pi \in \Pi$ occurs more than once in T, then $\exists i$, such that $\texttt{prev}(T[i \dots n]) \neq \texttt{prev}(T)[i \dots n]$, $1 \leqslant i \leqslant n$.*

Proof. Consider that the *only* parameter symbol to occur in the p-string T is $\pi \in \Pi$, which exists *only* at positions α and β with $\alpha < \beta$. Suppose that indeed $\texttt{prev}(T[\alpha \dots n]) = \texttt{prev}(T)[\alpha \dots n]$ and $\texttt{prev}(T[\beta \dots n]) = \texttt{prev}(T)[\beta \dots n]$. By Definition 3, the first occurrence of symbol π at position α will be $\texttt{prev}$ encoded by 0 and the π at position β will be $\texttt{prev}$ encoded by $\beta - \alpha$. So, in the case of suffix α, $\texttt{prev}(T[\alpha \dots n]) = \texttt{prev}(T)[\alpha \dots n]$. At suffix β, the encoding of π at position β in T will *change* to 0 in $\texttt{prev}(T[\beta \dots n])$ by Definition 3 whereas $\texttt{prev}(T)[\beta \dots n]$ will retain the *old* encoding of $\beta - \alpha$ since symbol π *still* occurs in $\texttt{prev}(T)$ at position α. The π at position β forces $\texttt{prev}(T[\beta \dots n]) \neq \texttt{prev}(T)[\beta \dots n]$, which is a contradiction. □

It is tempting to concatenate all of the p-suffixes of an n-length text T, perform a standard *SA* construction, and post-process the results to construct the *pSA*. However, there are $O(n^2)$ symbols in all of the p-suffixes of T and so, the time and space of such an *SA* construction would have a complexity no better than quadratic in the length of T. Since there are currently few direct *pSA* constructions in the literature [12,16] and since theoretical *pSA* construction improvements are necessary before applications can consider using the *pSA* over the *pST*, we are motivated to offer different approaches to direct *pSA* construction. In this paper, we are influenced by the use of fingerprints in pattern matching [20] and the use of arithmetic coding in [2] to address standard suffix sorting to use these techniques *to assist* with the new challenges of *pSA* construction.

5. p-Suffix sorting via fingerprints

Our idea is to modify the traditional Karp and Rabin (KR) fingerprinting scheme presented in [15,28,20] to accommodate the *changing* nature of p-suffixes. The KR algorithm generates an integral *KR* "signature" or "fingerprint" code to represent a string using the lexicographical ordering of symbols [20]. By representing p-suffixes of the n-length text T through numeric fingerprints, we devise a mechanism to retain a "tangible" copy of the *changing* p-suffixes under the $\texttt{prev}$ encoding. In this section, we assume that n is not too large. That is, the *KR* codes can fit into standard integer representations such as **long long integer**. Further, it is assumed that we can perform operations and computations on an integral data type in constant time. In traditional *KR* fingerprinting [15,28,20], the modulus operator is used to force all fingerprints to be represented by an integer data type, causing collisions between equal signatures and non-equal patterns that are represented. In this section, we do not use the modulus operator on fingerprints because it will destroy the lexicographical orderings of the represented patterns. As a result of this limitation, we use this section as a stepping stone to provide ideas and basic theory used in our core contribution: p-suffix sorting with arithmetic coding.

We now denote the following variables that are continually reused throughout this section for the working p-string T of length n: $prevT = \texttt{prev}(T)$, $forwT = \texttt{forw}(T)$, $max = \texttt{maxdist}(prevT)$ (see below), $R = |\Sigma| + max + 2$. Throughout this work, we use symbols from the variable $prevT$ to efficiently compute symbols of the p-suffix encoding $\texttt{prev}(T[i \dots n])$ for any i, $1 \leqslant i \leqslant n$. Our fingerprinting approach relies on a lexicographical ordering implementation of Definition 4 to appropriately accommodate the $\texttt{prev}$ alphabet $\Sigma \cup \mathbb{Z} \cup \{\$\}$. Our ordering scheme, function $\texttt{map}$, is formalized in Definition 10.

Definition 10 *(Mapping function).* Let $max = \texttt{maxdist}(prevT) = \max\{prevT[i] \mid prevT[i] \in \mathbb{Z} \text{ for } 1 \leqslant i \leqslant n\}$. Let function $\alpha_i(x, X)$ return the lexicographical order $(1, 2, \dots, |X|)$ of the symbol x in alphabet X. We then define the function

Table 2
Lexicographical ordering of p-suffixes with pKR, using $T = AwBzABwz\$$.

i	$pSA[i]$	$T[pSA[i]\ldots n]$	$\texttt{prev}(T[pSA[i]\ldots n])$	$\texttt{pKR}(pSA[i])$	$\texttt{KR}(pSA[i])$
1	9	$	$	43 046 721	43 046 721
2	8	z	0$	90 876 411	263 063 295
3	7	wz	00$	96 190 821	330 556 302
4	4	$zABwz$	0AB04$	129 298 356	129 593 601
5	2	$wBzABwz$	0B0AB54$	130 740 084	130 740 084
6	1	$AwBzABwz$	A0B0AB54$	358 900 444	358 900 444
7	5	$ABwz$	AB00$	388 608 030	391 501 431
8	6	Bwz	B00$	398 108 358	424 148 967
9	3	$BzABwz$	B0AB04$	401 786 973	401 819 778

$\texttt{map} : (\Sigma \cup \mathbb{Z} \cup \{\$\}) \to \mathbb{N}$ to map a symbol, say x, in *prevT* to an integer preserving the ordering of Definition 4. We also define the function $\texttt{in}(x, X)$ to determine if $x \in X$ instantaneously based on the definition of $\texttt{map}(x)$.

$$\texttt{map}(x) = \begin{cases} 1, & \textbf{if } x = \$ \\ \alpha_i(x, \mathbb{Z}) + 1, & \textbf{if } x \in \mathbb{Z} \\ \alpha_i(x, \Sigma) + max + 2, & \textbf{if } x \in \Sigma \end{cases}$$

$$\texttt{in}(x, X) = \begin{cases} true, & \textbf{if } X = \mathbb{Z} \wedge (1 < \texttt{map}(x) \leqslant max + 2) \\ true, & \textbf{if } X = (\Sigma \cup \{\$\}) \wedge (\texttt{map}(x) = 1 \vee \texttt{map}(x) > max + 2) \\ false, & \textbf{otherwise} \end{cases}$$

In practice, we can further tailor the map function for each single prev encoding by replacing each $\mathbb{Z}$ with a new set $\mathbb{Z}_{\texttt{prev}(T)}$ that specifies only the distances used in $\texttt{prev}(T)$. The function map is fundamental for the parameterized Karp–Rabin fingerprinting (pKR) algorithm, which generates parameterized Karp–Rabin (*pKR*) codes.

Definition 11 *(Parameterized Karp–Rabin (pKR) function).* Let $prevT_i = \texttt{prev}(T[i\ldots n])$. We define $\texttt{pKR}(i) = \sum_{k=n}^{i}[R^{k-1} \times \texttt{map}(prevT_i[n-k+1])]$ to return a fingerprint generated for the p-suffix beginning at index i.

Table 2 shows example fingerprints using our pKR algorithm and also the standard algorithm KR for the string $T = AwBzABwz\$$. This example shows the true power of our pKR algorithm in that the ordering of the computed fingerprints for p-suffixes of T yields the correct p-suffix array $pSA = \{9, 8, 7, 4, 2, 1, 5, 6, 3\}$. We notice that using KR directly produces the array $\{1, 4, 5, 2, 3, 6, 7, 9, 8\}$, which is not the correct p-suffix array. The execution of function pKR may be *naïvely* cascaded to produce fingerprints for all n p-suffixes at positions $1 \leqslant i \leqslant n$ of p-string T requiring $O(n^2)$ time, which is a theoretical bottleneck. We can intelligently construct *pKR* codes for the p-suffixes of T by taking advantage of the relationship between p-suffixes and *pKR* codes. Consider q_i to be the *pKR* code for the p-suffix at position i. The code q_{i+1} can be used to compute the fingerprint for q_i for $i \geqslant 1$ by introducing a new symbol at position i. Lemmas 12 and 13 identify the adjustments that dynamically change the p-suffixes between the neighboring p-suffixes at i and $(i+1)$ when considering a symbol introduced at position i.

Lemma 12. *Given p-string T, $prevT = \texttt{prev}(T)$, and $prevT[i+1\ldots n] = \texttt{prev}(T[i+1\ldots n])$ where $T[i]$ is a constant, terminal, or the* only occurrence *of parameter $T[i]$ in $T[i\ldots n]$, then $prevT[i\ldots n] = \texttt{prev}(T[i\ldots n])$ if $prevT[i] = \texttt{prev}(T[i])$.*

Proof. For symbol $\sigma \in (\Sigma \cup \{\$\})$, $\texttt{prev}(\sigma) = \sigma$ by Definition 3. For symbol $\pi \in \Pi$ Definition 3 states that $\texttt{prev}(\pi) = 0$ for the first occurrence. When $T[i]$ is the only occurrence of π in $T[i\ldots n]$, $\exists$ no future π to re-encode at positions $i+1$ to n by Definition 3. Since we are given that $prevT[i+1\ldots n] = \texttt{prev}(T[i+1\ldots n])$, and Definition 3 states that σ or π will generate a definitive encoding without impacting current encodings, then $prevT[i\ldots n] = \texttt{prev}(T[i\ldots n])$ upon adjustment of the encoding at $prevT[i]$ so that $prevT[i] = \texttt{prev}(T[i])$. □

Lemma 13. *Given p-string T, $prevT = \texttt{prev}(T)$, $forwT = \texttt{forw}(T)$, and $prevT[i+1\ldots n] = \texttt{prev}(T[i+1\ldots n])$ where $T[i] \in \Pi$ occurs multiple times in $T[i\ldots n]$, then $prevT[i\ldots n] = \texttt{prev}(T[i\ldots n])$ after (1) identifying the current parameter as the first occurrence of $T[i]$ ($prevT[i] = 0$) and (2) modifying the future occurrence of $T[i]$ ($prevT[i + forwT[i]] = forwT[i]$).*

Proof. We must achieve $\texttt{prev}(T[i\ldots n])$ by using $prevT[i\ldots n]$ given that $prevT[i+1\ldots n]$ is the correct p-suffix for position $(i+1)$. Since $T[i] \in \Pi$ is the *first* occurrence of $T[i]$ in $T[i\ldots n]$, by Definition 3, its encoding is clearly $\texttt{prev}(T[i]) = 0$. So, $prevT[i] = 0$ will adjust our p-suffix. However, since we are given the fact that $T[i]$ has future occurrences in $T[i+1\ldots n]$, then $\exists$ exactly one future occurrence of $T[i]$ to adjust. This occurrence of $T[i]$ in $T[i+1\ldots n]$ at position, say j, $j > i$ is currently such that $prevT[j] = 0$ and by Definition 3, *only* the first occurrence of a $T[i]$ in $\texttt{prev}(T[i\ldots n])$ can be 0. Then, clearly Definition 3 states that the encoding $prevT[j] = j - i$. To make this change we must locate the next forward

Algorithm 1. p-Suffix sorting with fingerprints.

```
1  struct pcode { int i, long long int pKR }
2  int[] p_suffix_sort_pKR(char T[n]) {
3      pcode code[n], long long int pKR=0
4      int pSA[n], k
5      // A) -- generate the individual prev fingerprints
6      for (k=n to 1) {
7          pKR = δ_pKR(k,pKR)
8          code[k]=(k,pKR)
9      } // B) -- sort p-suffixes
10     // sort code[k].pKR into code, 1≤k≤n
11     radix_sort({code[1],code[2], ... ,code[n]})
12     // C) -- retain p-suffix array
13     for (k=1 to n)
14         pSA[k]=code[k].i
15     return pSA
16 }
```

parameter $T[i]$ in $T[i+1 \ldots n]$, which Definition 8 informs us is available at $forwT[i]$ positions ahead of the current symbol position i; i.e. $j = i + forwT[i]$. So, $prevT[j] = j - i$ must be the case. By substituting j, $prevT[i + forwT[i]] = (i + forwT[i]) - i \Rightarrow prevT[i + forwT[i]] = forwT[i]$. □

We refer to a code generated by pKR for the p-suffix i as q_i. The transitions needed to compute a p-suffix i from a p-suffix $(i+1)$ formalized in Lemmas 12 and 13 are subsequently the requirements to compute code q_i from q_{i+1}. These transitions are consolidated into δ_{pKR} and shown to efficiently generate *pKR* codes.

Definition 14 *(Function δ_{pKR}).* Let $\beta = forwT[i]$, $\lambda = (\mathtt{map}(\beta) - \mathtt{map}(0)) \times R^{n-\beta-1}$, and $B = \frac{q_{i+1} + \mathtt{map}(\mathtt{prev}(T[i]))R^n}{R}$. We define the function $\delta_{\mathrm{pKR}}(i, q_{i+1})$ as follows to return the code q_i via a transition of the provided code q_{i+1} with the newly added symbol at position i.

$$\delta_{\mathrm{pKR}}(i, q_{i+1}) = \begin{cases} B, & \textbf{if } \mathtt{in}(prevT[i], \Sigma \cup \{\$\}) \vee (\mathtt{in}(prevT[i], \mathbb{Z}) \wedge forwT[i] \geqslant n) \\ B + \lambda, & \textbf{if } \mathtt{in}(prevT[i], \mathbb{Z}) \wedge forwT[i] < n \end{cases}$$

Theorem 15. *Given a p-string T of length n and precalculated variables prevT and forwT, function δ_{pKR} requires $O(n)$ time to generate fingerprints for all p-suffixes in T.*

Proof. The fingerprints are generated successively by the function calls $q_n = \delta_{\mathrm{pKR}}(n, 0)$, $q_{n-1} = \delta_{\mathrm{pKR}}(n-1, q_n), \ldots, q_1 = \delta_{\mathrm{pKR}}(1, q_2)$. Either case of function δ_{pKR} may be computed in $O(1)$ time and is called sequentially a total of n times, once for each of the n p-suffixes. The overall time is $O(n)$. □

We introduce p_suffix_sort_pKR in Algorithm 1 to sort p-suffixes via the sorting of fingerprints through the transition function in Definition 14. Theorem 16 proves the time complexity of Algorithm 1.

Theorem 16. *Given a p-string T of length n, function* p_suffix_sort_pKR *sorts all the n p-suffixes of T in $O(n)$ time.*

Proof. We assume that the fingerprints for each p-suffix are represented by an integer code and each use of the code is accomplished in constant time. Thus, section A) of p_suffix_sort_pKR follows from Theorem 15 to require $O(n)$ time. The radix sorting required in section B) requires $O(cn)$, where c is a constant. The loop in section C) clearly requires $O(n)$ time. Overall, p_suffix_sort_pKR requires $O(n)$ time. □

Again, we stress that this algorithm requires fingerprints that have not been processed via the modulus operation as was used in [20]. We need the lexicographical ordering of the p-suffixes to be preserved in the fingerprint ordering. Thus, the algorithm is suitable for small strings. Even with this limitation, the process of developing p_suffix_sort_pKR has introduced ideas and theory that are relevant to the main contribution of this work: p-suffix sorting with arithmetic codes.

6. p-Suffix sorting via arithmetic coding

Arithmetic coding compresses a string by recursively dividing up a real number line into intervals that account for the cumulative distribution function (*cdf*), which describes the probability space of each symbol. Let the pair (lo, hi) denote the interval for an arithmetic code *AC*, where *lo* and *hi* are the low and high boundaries, respectively. Any consistent

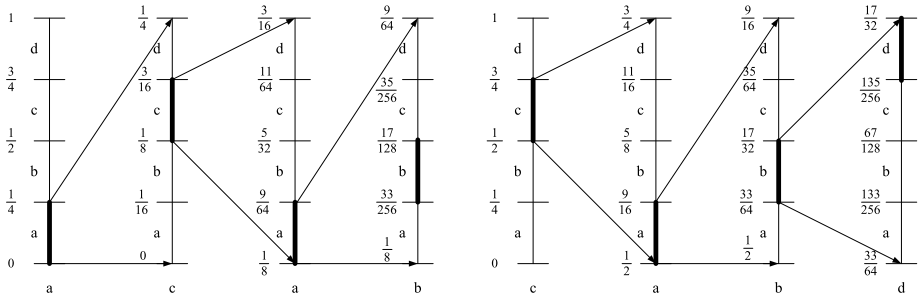

Fig. 1. Transitioning the *AC m-block* code from $\boxed{a}cab \rightarrow cab \rightarrow cab\boxed{d}$.

Algorithm 2. Generating *pAC* for an m-length prefix of p-suffix i.

```
1  struct AC { long double lo, long double hi }
2  AC pAC(int i, int m) {
3      int end=min{i+m−1,n}, k
4      int prevT_i[] = prev(T[i...end]), AC new=(0,0), old=(0,1)
5      for (k=i to end) {
6          new.hi=old.lo+(old.hi−old.lo)∗cdf[map(prevT_i[k−i+1])]
7          new.lo=old.lo+(old.hi−old.lo)∗cdf[map(prevT_i[k−i+1])−1]
8          old=new
9      }return new
10 }
```

choice in this region, say $\texttt{tag}(s) = \frac{s.hi+s.lo}{2}$, represents the arithmetic code and preserves the lexicographical ordering of strings. Arithmetic coding and renormalization are further described in [26,11]. Recently, Adjeroh and Nan [2] used a novel application of Shannon–Fano–Elias codes from information theory to address the traditional suffix sorting problem. In the work, they generate arithmetic codes for *m-blocks*, or m-length prefixes of the suffixes, to maintain the ordering of m symbols. They show how to efficiently transition one *AC m-block* code at suffix i to construct the *m-block AC* at suffix $(i+1)$ by removing the symbol at i and appending the symbol at $(i+m)$. The transitioning scheme is illustrated in Fig. 1. In terms of suffix sorting with arithmetic codes in [2], the suffixes are recursively partitioned and the generated *m-block* arithmetic codes are exploited to induce the ordering of the partitions in linear time. Extending the suffix sorting via arithmetic coding algorithm given in [2] to the p-suffix sorting problem is not straightforward because of the dynamic relationship between p-suffixes, identified in Lemma 9. In this section, we use arithmetic coding *to assist* in *pSA* construction. It is assumed that we can perform operations and computations on integral and floating point data types in constant time.

6.1. Sorting p-suffixes on average

Given an n-length p-string T, we can create a parameterized arithmetic code *pAC* via function pAC from Definition 17 for the *m-blocks*, or m-length prefixes, of the n p-suffixes of T. The distribution of symbols will impact the size of the intervals and hence the tag, but this does not change the order of the generated arithmetic codes. Thus, without loss of generality, we assume that each symbol $x \in (\Sigma \cup \mathbb{Z} \cup \{\$\})$ in the alphabet of a prev encoding is equally probable, where p represents the probability of a symbol and the array *cdf* contains the values of the uniform cdf with respect to the neighboring lexicographical alphabet symbols. The following definition modifies the traditional AC algorithm to create an *m-block* arithmetic code for a p-suffix at position i in T.

Definition 17 *(Parameterized arithmetic coding (*pAC*) function).* For an n-length p-string T, the function pAC in Algorithm 2 will generate an arithmetic code interval for the m-block prefix of the p-suffix starting at position i.

Table 3 shows the *pAC* codes for *m-blocks* of $m = 2, 3, n$ of p-string $T = AwBzABwz\$$. We notice that a "collision" occurs for two *pAC* codes using $m = 2$ since the *m-blocks* are equivalent. Even though the *pAC* codes distinctly sort the n p-suffixes of T when m approaches n, we highlight that the practical limitation is arithmetic precision. See [2,26] for handling this problem.

In order to use the m-block codes, we must generate them efficiently. We denote the m-block arithmetic code at p-suffix i by pAC_i. The idea is to first use function pAC to compute pAC_1 and use this code to generate the remaining $(n-1)$ codes, namely $pAC_2, pAC_3, \ldots,$ and pAC_n. Iteratively, we will need to adjust the arithmetic codes to (1) remove the old symbol and (2) add the new symbol. Recall from Lemmas 12 and 13 that even though a p-suffix is dynamically changing with respect to the starting position of a parameter, it is possible to transition or change, in a constant number of steps, a p-suffix to its

Table 3
Lexicographical ordering of p-suffixes with pAC, using $T = AwBzABwz\$$.

i	$pSA[i]$	$T[pSA[i]\ldots n]$	$\texttt{prev}(T[pSA[i]\ldots n])$	$\texttt{tag}(\texttt{pAC}(pSA[i], m))$		
				$m=2$	$m=3$	$m=n$
1	9	\$	\$	0.055556	0.055556	0.055556
2	8	$z\$$	$0\$$	0.117284	0.117284	0.117284
3	7	$wz\$$	$00\$$	0.129630	0.124143	0.124143
4	4	$zABwz\$$	$0AB04\$$	0.203704	0.209191	0.208743
5	2	$wBzABwz\$$	$0B0AB54\$$	0.216049	0.211934	0.212459
6	1	$AwBzABwz\$$	$A0B0AB54\$$	0.796296	0.801783	0.801384
7	5	$ABwz\$$	$AB00\$$	0.882716	0.878601	0.878076
8	6	$Bwz\$$	$B00\$$	0.907407	0.903292	0.902683
9	3	$BzABwz\$$	$B0AB04\$$	0.907407	0.911523	0.912083

neighboring p-suffix in the p-string. This transition idea was used in the construction of the *pKR* codes in Definition 14. We now use this transition idea to construct *pAC* codes.

Case 1. *Removing a symbol s from an arithmetic code m-block requires us to simply delete s when $s \in \Sigma$ or $s \in \Pi$ and does not occur in the m-block. When $s \in \Pi$ and occurs later in the m-block, the code must accommodate for both the removed occurrence and the future occurrence of s.*

Definition 18 *(Remove symbol δ^-_{pAC} transition).* Given the AC code A at m-*block* i with $1 \leqslant i \leqslant n$, δ^-_{pAC} supplies the transition to remove the symbol at position i and provide the new code A of the $(m-1)$-*block* at p-suffix $(i+1)$. Let $\beta = forwT[i]$, $j = i+\beta$, $e = \min\{i+m-1, n\}$, $\lambda = (\texttt{map}(\beta) - \texttt{map}(0)) \times p^{\beta+1}$, and $c = cdf[\texttt{map}(\texttt{prev}(T[i])) - 1]$.

$$\delta^-_{\text{pAC}}(i, A) = \begin{cases} \left(\frac{A.lo-c}{p}, \frac{A.hi-c}{p}\right), & \textbf{if } (\texttt{in}(prevT[i], \mathbb{Z}) \wedge j > e) \vee \texttt{in}(prevT[i], \Sigma \cup \{\$\}) \\ \left(\frac{A.lo-\lambda-c}{p}, \frac{A.hi-\lambda-c}{p}\right), & \textbf{if } \texttt{in}(prevT[i], \mathbb{Z}) \wedge j \leqslant e \end{cases}$$

Case 2. *Adding (i.e. appending) symbol s to an arithmetic code m-block requires us to simply append the code when $s \in \Sigma$ or $s \in \Pi$ and does not occur in the m-block. When $s \in \Pi$ and occurs previously in the m-block, the code must account for the new occurrence in terms of the previous occurrence of s.*

Definition 19 *(Add symbol δ^+_{pAC} transition).* Given the AC code A at $(m-1)$-*block* $(i-m+1) \geqslant 1$ with $i \leqslant n$, δ^+_{pAC} supplies the transition to add the symbol at position i and provide the new code A of the m-*block* at p-suffix $(i-m+1)$. Let $b = \max\{1, i-m+1\}$, $k = i - prevT[i]$, $\Delta = A.hi - A.lo$, $d = \Delta \times cdf[\texttt{map}(\texttt{prev}(T[i]))]$, $f = \Delta \times cdf[\texttt{map}(\texttt{prev}(T[i])) - 1]$, $v = \Delta \times cdf[\texttt{map}(prevT[i])]$, and $w = \Delta \times cdf[\texttt{map}(prevT[i]) - 1]$

$$\delta^+_{\text{pAC}}(i, A) = \begin{cases} (A.lo + f, A.lo + d), & \textbf{if } (\texttt{in}(prevT[i], \mathbb{Z}) \wedge k < b) \vee \texttt{in}(prevT[i], \Sigma \cup \{\$\}) \\ (A.lo + w, A.lo + v), & \textbf{if } \texttt{in}(prevT[i], \mathbb{Z}) \wedge k \geqslant b \end{cases}$$

With the assistance of Definitions 18 and 19, we can efficiently generate the m-*block* codes for all n p-suffixes of T. Consider the p-string $T = zwzAB A\$$, $\Sigma = \{A, B\}$, $\Pi = \{w, z\}$, and $m = 4$, we successively generate the m-*block* codes in the following fashion: $\boxed{0}0\boxed{2}A \overset{\delta^-_{\text{pAC}}}{\rightarrow} 00A \overset{\delta^+_{\text{pAC}}}{\rightarrow} 00A\boxed{B} \rightarrow \cdots$.

Theorem 20. *Given a p-string T of length n and precalculated variables prevT and forwT, the pAC codes for all the m-length prefixes of the p-suffixes require $O(n)$ time to generate.*

Proof. Generating the first m-block code pAC_1 via $pAC_1 = \texttt{pAC}(1, m)$ will require $O(m)$ time. Iteratively, the neighboring *pAC* codes will be used to create the successive p-suffix codes. The first extension of code pAC_1 to create pAC_2 will require the removal of $prevT[1]$ via a call to $pAC_2 = \delta^-_{\text{pAC}}(1, pAC_1)$, which is $O(1)$ work, and the addition of symbol $prevT[2+m-1]$ via a call to $pAC_2 = \delta^+_{\text{pAC}}(2+m-1, pAC_2)$, which also demands $O(1)$ work. This process requiring two $O(1)$ steps is needed for the remaining $(n-1)$ m-block p-suffixes of T. The resulting time is $O(m+n)$. Since $m \leqslant n$, the theorem holds. □

The efficient preprocessing from Theorem 20 leads to an average case linear time algorithm for direct p-suffix sorting for non-binary parameter alphabets. This result is a significant checkpoint in the paper since it is later extended to sort p-suffixes in the worst case.

Theorem 21. *Given a p-string T of length n, p-suffix-sorting of T can be accomplished in $O(n)$ time and $O(n)$ space on average via parameterized arithmetic coding.*

Algorithm 3. Average case p-suffix sorting with arithmetic codes.

```
1  struct pcode { int i, int d_i }
2  boolean average_case=true
3  int[] pSA_Algorithm3(char T[n]) {
4      int pSA[n], m=⌈log(n)⌉, k
5      pcode code[n], AC current=pAC(1,min{m,n})
6      for (k=2 to n) {
7          current=δ⁻_pAC(k−1,current)
8          if (k+m−1 ⩽ n)
9              current=δ⁺_pAC(k+m−1,current)
10         code[k]=(k,⌊c∗(n−1)∗(tag(current)−tag(pAC_min))/(tag(pAC_max)−tag(pAC_min))⌋)
11     } // sort code[k].d_i into code, 1⩽k⩽n
12     radix_sort({code[1],code[2],...,code[n]})
13     for (k=1 to n) {
14         pSA[k]=code[k].i
15         if (k<n ∧ code[k].d_i=code[k+1].d_i)
16             average_case=false
17     } return pSA
18 }
```

Proof. We can construct $\texttt{prev}(T)$ in $O(n)$ time given an indexed alphabet and an $O(|\Pi|)$ auxiliary data structure. The lexicographical ordering of the m-block pAC codes follow from the notion of arithmetic coding and Definition 10. From Theorem 20, we can create all the m-block $\texttt{pAC}$ codes in $O(n)$ time. Similar to [2], the individual floating point codes may be converted to integer codes d_i in the range $[0, c(n-1)]$ by $d_i = \lfloor c(n-1)\frac{\texttt{tag}(pAC_i)-\texttt{tag}(pAC_{min})}{\texttt{tag}(pAC_{max})-\texttt{tag}(pAC_{min})}\rfloor$, where the constant $c > 1$ is chosen to best separate the d_i and values pAC_{min} and pAC_{max} correspond to the minimum and maximum pAC codes, respectively. From [19,13], we know that an n-length general (average) string has a max length for the longest common prefix (i.e. maximum value in the *LCP* array) in $O(\log_{|\Sigma|} n)$. That is, a standard average string from alphabet Σ has prefixes that match $O(\log_{|\Sigma|} n)$ symbols until the suffixes clearly differentiate. We can use this result in terms of p-strings by converting the dynamically changing p-suffixes of T to a single standard string. Let $x \circ y$ be the string concatenation of x and y. Then, we shall make a standard string $Q = \texttt{prev}(T[1 \ldots n-1])\$_1 \circ \texttt{prev}(T[2 \ldots n-1])\$_2 \circ \cdots \circ \texttt{prev}(T[n-1 \ldots n-1])\$_{n-1} \circ \$_n$ that contains each individual p-suffix of T. Notice that Q is of length $|Q| = \frac{n(n+1)}{2} \in O(n^2)$ and since all p-suffixes are clearly represented, the symbols of Q may be mapped to the standard string alphabet Σ. Since there are at most n unique non-terminal symbols used by all of the p-suffixes, it is possible that $|\Sigma| = n$. Now, we shall use the contribution of [19,13] to obtain the length of the maximum longest common prefix for the average string Q, which will be in $O(\log_{|\Sigma|} n^2)$. Thus, we can upper bound the result by $O(\log n)$. Then by choosing $m = O(\log n)$ and generating the m-block pAC codes, only the first $O(n)$ radix sort of the d_i codes is required to differentiate the p-suffixes of an average case string, demanding only $O(n)$ operations. Such is accomplished in Algorithm 3. Since the algorithm only uses a constant number of arrays of length n, then only $O(n)$ space is consumed. □

6.2. Sorting p-suffixes in the worst case

Given a general p-string, the p-suffix sorting approach in Algorithm 3 sorts p-suffixes. A limitation of the average case solution in Algorithm 3 is that it does not support p-suffix sorting in the worst case. More specifically, when the $m = O(\log n)$ symbols encoded by the m-block arithmetic codes are not sufficient to differentiate between all of the n p-suffixes of a p-string, additional sorting is required. At this point, we have only referred to an m-block as an m-length prefix of some p-suffix. In general, an m-block can be any m-length substring of a p-suffix. Thus, we use the m-block idea to extend Algorithm 3 to sort the n p-suffixes of a p-string in the worst case by successively sorting general m-block arithmetic codes.

Consider the n-length p-string T and some m, $1 \leqslant m \leqslant n$. Our idea is to first construct the m-block arithmetic codes $\texttt{prev}(T[1 \ldots m]), \texttt{prev}(T[2 \ldots m+1]), \ldots, \texttt{prev}(T[n-m+1, n]), \ldots, \texttt{prev}(T[n \ldots n])$ and then sort the codes to form the initial buckets. While each individual bucket is not a singleton, the buckets are subsequently ordered within by sorting the arithmetic codes corresponding to the adjacent m-block p-suffixes $\texttt{prev}(T[1 \ldots n])[m+1 \ldots 2m], \texttt{prev}(T[2 \ldots n])[m+1 \ldots 2m]$, etc. to further differentiate the p-suffixes within each individual bucket. This process continues until each bucket is a singleton; that is, there are n buckets, each containing only one p-suffix. We further pursue this approach.

Since we are relaxing the location of m-blocks in a p-suffix, the sorting technique described previously will require us to extend Algorithm 3 by first generalizing our use of m-block codes so that m-blocks apply to *any* m-length substring of a p-suffix. Like Algorithm 3, we want to efficiently generate arithmetic codes by constructing the first m-block code for the p-suffix at index 1 and then, extending the codes for p-suffixes $2, 3, \ldots, n$ with transition functions. We first generalize the $\texttt{pAC}$ function in Definition 22.

Algorithm 4. Generalized algorithm to generate a *pAC* code.

```
1  struct AC { long double lo, long double hi }
2  AC pAC(char prevStr[]) {
3      int end=|prevStr|, k
4      AC new=(0,0), old=(0,1)
5      for (k=1 to end) {
6          new.hi=old.lo+(old.hi−old.lo)∗cdf[map(prevStr[k])]
7          new.lo=old.lo+(old.hi−old.lo)∗cdf[map(prevStr[k])−1]
8          old=new
9      } return new
10 }
```

Definition 22 *(Generalized parameterized arithmetic coding (*pAC*) function).* For an n-length p-string T, the function pAC in Algorithm 4 will generate an arithmetic code interval for a given substring *prevStr* from some prev encoding.

Next, the transition functions are generalized in Definitions 23 and 24 to account for any m-block of a p-suffix. The generalized transition functions that consider symbols with equal probability p are presented below and are similar in nature to those presented in Definitions 18 and 19 (which only work for prefixes of p-suffixes). The new transitions differ primarily by incorporating the index q as a parameter to indicate the p-suffix containing the m-block.

Definition 23 *(Remove symbol δ^-_{pAC} generalized transition).* Given the AC code A at the m-block beginning at $i-q+1 \geqslant 1$ in p-suffix $q \geqslant 1$, δ^-_{pAC} supplies the transition to remove the symbol at position i in *prevT* and provide the new code A of the $(m-1)$-block at p-suffix $q+1$. Let $r = i$, $\beta = forwT[q]$, $j = q+\beta$, $e = \min\{i+m-1, n\}$, $\lambda = (\mathtt{map}(\beta) - \mathtt{map}(0)) \times p^{\beta\%m+1}$, $v = prevT[r]$ iff $(\mathtt{in}(prevT[r], \Sigma \cup \{\$\}) \vee (\mathtt{in}(prevT[r], \mathbb{Z}) \wedge r - prevT[r] \geqslant q))$ and otherwise, $v = 0$. Finally, let $c = cdf[\mathtt{map}(v) - 1]$.

$$\delta^-_{\mathrm{pAC}}(q, i, A) = \begin{cases} \left(\frac{A.lo-c}{p}, \frac{A.hi-c}{p}\right), & \textbf{if } (\mathtt{in}(prevT[q], \mathbb{Z}) \wedge (j > e \vee j \leqslant r)) \vee \mathtt{in}(prevT[q], \Sigma \cup \{\$\}) \\ \left(\frac{A.lo-\lambda-c}{p}, \frac{A.hi-\lambda-c}{p}\right), & \textbf{if } \mathtt{in}(prevT[q], \mathbb{Z}) \wedge r < j \leqslant e \end{cases}$$

Definition 24 *(Add symbol δ^+_{pAC} generalized transition).* Given the AC code A at *$(m-1)$-block* $(i-m+1) \geqslant 1$ with $i \leqslant n$ of p-suffix $q \geqslant 1$, δ^+_{pAC} supplies the transition to add the symbol at position i in *prevT* and provide the new code A of the *m-block* in p-suffix q. Let $a = i$, $k = a - prevT[a]$, $\Delta = A.hi - A.lo$, $d = \Delta \times cdf[\mathtt{map}(\mathtt{prev}(T[a]))]$, $f = \Delta \times cdf[\mathtt{map}(\mathtt{prev}(T[a])) - 1]$, $v = \Delta \times cdf[\mathtt{map}(prevT[a])]$, and $w = \Delta \times cdf[\mathtt{map}(prevT[a]) - 1]$

$$\delta^+_{\mathrm{pAC}}(q, i, A) = \begin{cases} (A.lo + f, A.lo + d), & \textbf{if } (\mathtt{in}(prevT[a], \mathbb{Z}) \wedge k < q) \vee \mathtt{in}(prevT[a], \Sigma \cup \{\$\}) \\ (A.lo + w, A.lo + v), & \textbf{if } \mathtt{in}(prevT[a], \mathbb{Z}) \wedge k \geqslant q \end{cases}$$

Using the previously defined functions, the worst case p-suffix sorting approach is given in Algorithm 5. In the algorithm, each p-suffix is placed into an initial bucket based on the arithmetic codes *pAC* (in the range $0 \leqslant pAC \leqslant 1$) constructed from the initial m-blocks, that is, the m-blocks $\mathtt{prev}(T[1 \ldots m]), \mathtt{prev}(T[2 \ldots m+1])$, etc. The floating point data type is exploited to maintain the number of each bucket b and each *pAC* code in the form $\langle b.pAC \rangle$. The floating point codes are sorted by a traditional merge sort procedure to sort the codes and in turn, sort the p-suffix m-blocks. The process is repeated as the *pAC* codes are computed for the adjacent m-blocks of $\mathtt{prev}(T[1 \ldots n])[m+1 \ldots 2m], \mathtt{prev}(T[2 \ldots n])[m+1 \ldots 2m]$, etc. The computed codes are used jointly with the bucket identifiers to successively sort within each bucket. The buckets are subsequently renumbered to accommodate the current sorting. When there are n buckets, this signifies that each p-suffix is distinct to a bucket and hence, the sorting is complete. Moreover, the sorting is complete when we process enough m-blocks to clearly differentiate the p-suffixes or more precisely, h such m-blocks where

$$h = \begin{cases} \max\{1, u\}, & \textbf{if } n\%m \neq 0 \\ \max\{1, \min\{\lceil \frac{n}{m} \rceil, u+1\}\}, & \textbf{otherwise} \end{cases}$$

with $u = \lceil \frac{\mathtt{lcp}(T[i \ldots n], T[j \ldots n])}{m} \rceil$, $1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant n$, $i \neq j$. The running time of Algorithm 5 is proven in Theorem 25.

Theorem 25. *Given a p-string T of length n and an integer m with $1 \leqslant m \leqslant n$, the p-suffix-sorting of T can be accomplished in $O(\frac{n^2 \log n}{m})$ time in the worst case and $O(n \log n)$ on average via parameterized arithmetic coding.*

Proof. It is clear from Definition 4 and the lexicographical ordering preserved in an arithmetic code that the successive sorting within buckets in Algorithm 5 will correctly sort the n p-suffixes of T. In terms of time, section A) requires a total of $O(n)$ operations for initialization. Section B) performs the actual sorting and the bulk of the work. The steps in section B.1) are similar to the $O(n)$ steps in Theorem 20. In this chunk of the algorithm, the work is completed in $O(m+n) \in O(n)$

Algorithm 5. Worst case p-suffix sorting.

```
1  struct pcode { int i, long double f_i }
2  int[] pSA_Algorithm5(char T[n]) {
3      int pSA[n], R[n], k, q=1, b=0, s=1
4      int rmv, add
5      pcode code[n], tmp[n]
6      AC current
7      // A) -- initialize code pairs
8      for (k=1 to n) {
9          code[k]=tmp[k]=(k,0)
10         R[k]=k
11     } // B) -- sort m-blocks of p-suffixes
12     while (b < n) {
13         // B.1) -- generate m-block codes
14         current=pAC(prevT[q...min{q+m-1,n}])
15         tmp[R[1]]=(1,tag(current))
16         rmv=q, add=q+m
17         for (k=2 to n) {
18             if (1 ⩽ rmv ⩽ n)
19                 current=δ⁻_pAC(k,rmv,current)
20             if (1 ⩽ add ⩽ n)
21                 current=δ⁺_pAC(k+1,add,current)
22             tmp[R[k]]=(k,tag(current))
23             rmv++, add++
24         }
25         // B.2) -- identify bucket
26         for (k=1 to n) {
27             if (k=1)
28                 b=1
29             else {
30                 if (!(code[R[tmp[k].i]].f_i=code[R[tmp[k-1].i]].f_i))
31                     b++
32             } tmp[k].f_i=b+tmp[k].f_i-⌊tmp[k].f_i⌋
33         }
34         // B.3) -- sort p-suffixes
35         // sort f_i attribute of array tmp
36         merge_sort({tmp[1],tmp[2], ...,tmp[n]})
37
38         // B.4) -- rank the p-suffixes
39         for (k=1 to n)
40             R[tmp[k].i]=k
41         code=tmp
42         q=q+m
43         s++
44     }
45     // C) -- retain p-suffix array
46     for (k=1 to n)
47         pSA[k]=code[k].i
48     return pSA
49 }
```

time since $m \leqslant n$, the call to function $\mathtt{pAC}(S)$ (in Definition 22) demands $O(m)$ steps ($m = |S|$), and the transition functions $\delta^{-}_{\mathtt{pAC}}$ (in Definition 23) and $\delta^{+}_{\mathtt{pAC}}$ (in Definition 24) are clearly $O(1)$ operations. Sections B.2) and B.4) clearly use $O(n)$ operations. Section B.3) uses a traditional $O(n \log n)$ merge sort operation to sort n floating point numbers during each iteration. The operations in B.1) through B.4) are performed until $b \geqslant n$, i.e., until there is one distinct bucket for each p-suffix. More specifically, this occurs when we have iterated h times, processing each of the m-length codes reaching the $pLCP_{max}$ between any two p-suffixes. Thus, the $O(n + n \log n)$ operations in B.1) through B.4) are performed a total of $h = \lceil \frac{n}{m} \rceil$ times in the worst case. The retrieval of the p-suffix array in section C) only uses $O(n)$ operations. Hence, Algorithm 5 sorts the n p-suffixes of a p-string in $O(n + h \times (n + n \log n))$ time. Therefore, $O(h \times n \log n)$ time is required and in the worst case when $h = \lceil \frac{n}{m} \rceil$, $O(\frac{n^2 \log n}{m})$ time is demanded. Similar to the proof of Theorem 21, if we consider an average case, general p-string, to require one sort ($h = 1$) when $m = O(\log n)$ then the algorithm executes in $O(n \log n)$ on average. □

We acknowledge that it is possible to first call Algorithm 3 from Algorithm 5 to achieve an $O(n)$ average case result prior to worst case p-suffix sorting. Table 4 illustrates that based on m, the time complexity varies. In the table, we display various values for m and note that in practice, the choice of m should be made by considering the underlying architecture in terms of n and the alphabet size. For instance, consider a very large string of length $n = 1TB$ (assuming a 64-bit machine).

Table 4
Time and space complexity of Algorithm 5 ($n > 1$ and $m = \min\{f(n), n\}$).

$f(n)$	Time complexity ($\frac{n^2 \log n}{m}$)	Space complexity
1	$O(n^2 \log n)$	
$\lceil \log \log n \rceil$	$O(\frac{n^2 \log n}{\log \log n})$	
$\lceil \sqrt{\log n} \rceil$	$O(n^2 \sqrt{\log n})$	
$\lceil \log n \rceil$	$O(n^2)$	$O(n)$
$\lceil \log^2 n \rceil$	$O(\frac{n^2}{\log n})$	
$\lceil \sqrt[3]{n} \log n \rceil$	$O(n^{\frac{5}{3}})$	
$\lceil \frac{\sqrt{n}}{\log n} \rceil$	$O(n^{\frac{3}{2}} \log^2 n)$	
$\lceil \sqrt{n} \rceil$	$O(n^{\frac{3}{2}} \log n)$	

Table 5
Example execution of Algorithm 5 with $T = xxyxxyxx\$$ and $m = 2$, where $\diamond$ identifies a sorted bucket.

Sort $s = 1$			Sort $s = 2$			Sort $s = 3$			
b	m-block	p-suffix	b	m-block	p-suffix	b	m-block	p-suffix	$\texttt{prev}(T[b \ldots n])$
1	\$	9 $\diamond$	1		9	1		9	\$
2	0\$	8 $\diamond$	2		8	2		8	0\$
3	00	2	3	1\$	6 $\diamond$	3		6	001\$
		3	4	13	3 $\diamond$	4	21	3	001321\$
		5	5	21	2	5	\$	5 $\diamond$	0021\$
		6			5	6	32	2 $\diamond$	0021321\$
4	01	1	6	\$	7 $\diamond$	7		7	01\$
		4	7	02	1	8	1\$	4 $\diamond$	01021\$
		7			4	9	13	1 $\diamond$	01021321\$

Consider there to be limited number of unique symbols in $\texttt{prev}(T)$. Then, a practical choice is $m = \lceil \log n \rceil = 40$. Recall that in Algorithm 5, we use a whole number to represent the bucket number b and a fraction to represent the code pAC in the form $\langle b.pAC \rangle$. In practice, we can avoid some of the type-casting and calculations by dividing the b and pAC into two separate **long double** variables and handling them individually. The broad range of the **long double** data type can easily represent the pAC m-block codes since there are only n of them. With precision restrictions, we can increase the integral distance between each lexicographically ordered symbol by slightly altering the `map` function to further utilize the data type range for the pAC codes. Alternatively, it is possible to allocate each of the $\langle b.pAC \rangle$ elements by cascading a constant number of variables to address any data type limitations on very large data sets. Nonetheless, an architecture that is able to load $n = 1TB$ of data completely into main memory likely has the ability to allocate extra space and perform heavy computations to further process the large data set.

Moreover, a simple analysis of Algorithm 5 will show that only $O(n)$ space is required in any case. This is true because the algorithm maintains only a constant number of arrays containing data related to the n p-suffixes. Algorithm 5 is an improvement over the worst case p-suffix sorting in [16] with a theoretical time complexity of $O(n^2)$ because in our pSA construction, the tradeoff between m and time offers numerous cases with sub-quadratic worst case theoretical complexity.

An example execution of Algorithm 5 is provided in Table 5. In the example, notice that $n = 9$, $m = 2$, and in the worst case $s = \lceil \frac{n}{m} \rceil = \lceil \frac{9}{2} \rceil = 5$ sorts are needed, but only $s = 3$ sorts are used in this example. This is because the maximum m-blocks observed is $h = \max\{1, \lceil \frac{5}{2} \rceil\} = 3$ since the maximum `plcp` is $T[1 \ldots n] = 01021321\$ =_5 01021\$ = T[4 \ldots n]$. Algorithm 5 is designed to terminate early if possible, taking advantage of this very situation when only a few m-blocks are required to differentiate between the p-suffixes.

Further, we observe that during each iteration of Algorithm 5, it is guaranteed that m of the p-suffixes are in their final position in the pSA. The last m p-suffixes are guaranteed to be sorted because the size of the p-suffixes are $1, 2, \ldots, n$ and during each sorting iteration, exactly m of the smallest p-suffixes (also the last p-suffixes in the p-string) have no future symbols and thus, are fully differentiated. For the first iteration, these m p-suffixes are the indices $(n - m + 1)$ to n. Moreover, the p-suffixes $(n - 2m + 1)$ to $(n - m)$ are guaranteed to be sorted after the second iteration. We take advantage of this fact in Algorithm 6. In each iteration of section B) of Algorithm 6, the last m p-suffixes are not operated on. Instead, we alter the bucketing scheme to assign the last m p-suffixes to their final bucket (index) in the p-suffix array and thus, do not need to further process these p-suffixes. Theoretically, the time saved here is important because the sorting only needs to be performed until the p-suffixes differentiate and m less pAC codes are sorted each iteration. Also, we observe that our worst case p-suffix sorting routine must only be called when the p-suffixes of a particular p-string cannot be fully differentiated in the average case. So, we take advantage of this fact in Algorithm 6 by first calling our average case solution in Algorithm 3 and further sorting only if necessary. Theorem 26 proves the running time of Algorithm 6.

Algorithm 6. Improved worst case p-suffix sorting.

```
1  struct pcode { int i, long double f_i }
2  int[] pSA_Algorithm6(char T[n]) {
3      int pSA[n], R[n], R2[n], k, q=1, b=0, s=1
4      int i=0, f=0, x=n, y=0, rmv, add, v
5      pcode code[n], tmp[n], tmp2[n]
6      AC current
7      // A) -- init / average p-suffix sorting
8      for (k=1 to n) {
9          code[k]=tmp[k]=tmp2[k]=(k,0)
10         R[k]=R2[k]=k
11     } pSA=pSA_Algorithm3(T)
12     if (average_case=true) return pSA
13     // B) -- sort m-blocks of p-suffixes
14     while (f < x) {
15         // B.1) -- generate m-block codes
16         f=0, y=min{q+m-1,n}
17         current=pAC(prevT[q...y])
18         tmp[R[1]]=(1,tag(current))
19         rmv=q, add=q+m
20         for (k=2 to n) {
21             if (1 ⩽ rmv ⩽ n)
22                 current=δ⁻_pAC(k,rmv,current)
23             if (1 ⩽ add ⩽ n)
24                 current=δ⁺_pAC(k+1,add,current)
25             tmp[R[k]]=(k,tag(current)), rmv++, add++
26         }
27         // B.2) -- identify bucket
28         radix_sort({R2[1],R2[2], ... ,R2[x]})
29         for (k=1 to x) {
30             if (R2[k]=1){ b=1, f++
31             } else {
32                 if (!(code[R2[k]].f_i=code[R2[k]-1].f_i)){
33                     b=R2[k]+1, f++
34                 }
35             } tmp[R2[k]].f_i=b+tmp[R2[k]].f_i-⌊tmp[R2[k]].f_i⌋
36         } for (v=1 to x) tmp2[v]=tmp[R[v]]
37
38         // B.3) -- sort p-suffixes
39         // sort f_i attribute of array tmp2
40         merge_sort({tmp2[1],tmp2[2],... ,tmp2[x]})
41         for (v=1 to x) tmp[R2[v]]=tmp2[v]
42
43         // B.4) -- rank the p-suffixes
44         for (k=1 to x) R[tmp[R2[k]].i]=R2[k]
45         for (k=1 to x) R2[k]=R[k]
46         for (k=1 to x) code[R2[k]]=tmp[R2[k]]
47         q=q+m, i++, x=n-i*m, s++
48     }
49     // C) -- retain p-suffix array
50     for (k=1 to n) pSA[k]=code[k].i
51     return pSA
52 }
```

Theorem 26. *Given a p-string T of length n and an integer m with $1 \leqslant m \leqslant n$, the p-suffix-sorting of T is accomplished in $O(n)$ time on average and precisely $O(\sum_{i=1}^{h-1}[(n-i\times m)\log(n-i\times m)]+n\log n)$ time in the worst case via parameterized arithmetic coding. Worst case space is $O(n)$.*

Proof. The time complexity of Algorithm 6 is similar to the time analysis of Algorithm 5 in Theorem 25 except for two differences. First, the initial sorting of section A) may *completely* sort the p-suffixes via Algorithm 3 where $m = O(\log n)$ if and only if $h = O(\frac{\log n}{m}) = O(1)$. In this average case, Algorithm 6 performs a single iteration to sort the m-blocks of length $O(\log n)$ to sort a general p-string in $O(n)$ time by Theorem 21. Second, the sections B.1) through B.4) are called only when the sorting is incomplete, i.e. the p-suffixes require further sorting beyond the average case. These steps operate on m less elements during each iteration of section B). Let each individual $\{c_1, c_2\}$ be a constant that represents the time required to perform an operation and let c_3 be chosen to satisfy the inequality. Then, the time complexity is $c_1 \times \sum_{i=0}^{h-1}[(n-i\times m)\log(n-i\times m)+(n-i\times m)]+c_2\times n \leqslant c_3\times(\sum_{i=0}^{0}[n\log(n-i\times m)]+\sum_{i=1}^{h-1}[(n-i\times m)\log(n-i\times m)]) \in$

Table 6
Time and space complexity of Algorithm 6 ($n > 1$ and $m = \min\{f(n), n\}$).

h	$f(n)$	Time complexity	Space complexity
1	$\lceil \log n \rceil$	$O(n)$	$O(n)$
1	$\lceil \sqrt{n} \rceil$	$O(n)$	
$\lceil \log n \rceil$	$\lceil \log n \rceil$	$O(n \log n \log(n - \log n))$	
$\lceil \frac{n}{m} \rceil$	$\lceil \sqrt{n} \rceil$	$O(n^{\frac{3}{2}} \log(n - \sqrt{n}))$	
$\lceil \frac{n}{m} \rceil$	$\lceil \log n \rceil$	$O(\frac{n^2 \log(n - \log n)}{\log n})$	
$\lceil \frac{n}{m} \rceil$	1	$O(n^2 \log n)$	

$O(\sum_{i=1}^{h-1}[(n - i \times m) \log(n - i \times m)] + n \log n)$. Since Algorithm 6 only uses a constant number of arrays of length n, then $O(n)$ space is consumed in the worst case. □

The exact time complexity of Algorithm 6 is dependent on several factors including n, m, and h. This adds to the practical value of our algorithm, since n and h are dependent on the p-string and a practical m is chosen for arithmetic coding to control the overall time complexity. As mentioned earlier, the selection of m around $m = O(\log n)$ is a practical choice for large data sets. Note that the value of h is never individually computed and instead, the value of h is simply a precise measure for the running time of the algorithm. Examples of the running time of Algorithm 6 are given in Table 6. In addition to solving the sorting of sophisticated, dynamically changing p-suffixes of a p-string, our worst case algorithm also correctly sorts the suffixes of a traditional string, since a p-string without parameters behaves as a traditional string.

The running times displayed in Table 6 are dependent on the h for a provided p-string. In particular, the running times where $h = \lceil \frac{n}{m} \rceil$ address a p-string in the worst case. In [16], it was originally posed as an open question whether there exists a worst case p-suffix sorting approach that performs better than an $O(n^2)$ solution. More formally, does there exist a solution running in $o(n^2)$ time in the worst case? In the following corollary, we answer this open question in the affirmative. Yes, there are indeed $o(n^2)$ approaches to sort sophisticated, dynamically changing p-suffixes. Algorithm 6 accomplishes this feat.

Corollary 27. *Let $\epsilon > 0$ be a very small number $\epsilon = 0.00\ldots1$. For a p-string T of length n and a chosen $m = O(\log^{1+\epsilon} n)$, then in the worst case when $h = O(\frac{n}{m})$ the running time of Algorithm 6 is $o(n^2)$.*

Proof. From Theorem 26, Algorithm 6 executes in $g(n) = O(\frac{n^2 \log(n - \log^{1+\epsilon} n)}{\log^{1+\epsilon} n})$ time in the worst case. Since $\lim_{n\to\infty}[\frac{g(n)}{n^2}] = 0$, then also $g(n) \in o(n^2)$. □

7. Experiments

We implemented the newly introduced p-suffix array (*pSA*) construction algorithms based on our parameterized arithmetic coding (*pAC*) techniques. Our programs are written in C. We have executed our *pSA* constructions in the Cygwin environment running on a Dell Inspiron 570 desktop with 3.10 GHz clock speed and 8 GB RAM.

We tested on files from the Large Corpus (http://corpus.canterbury.ac.nz/descriptions/#large), the theoretical Fibonacci string, and strings from random distributions. Tables 7–10 show the nature of the dataset in terms of their average and maximum LCP values. For a given string, the mean and max LCP values give an idea on the difficulty in suffix sorting for the string, with higher values indicating more difficulty. At the time of this writing, we were unable to compare our programs with the only other worst case direct p-suffix sorting programs by [16] due to environment compilation issues. We discuss our experimental results and compare them with our theoretical results. When conducting these experiments, we select the alphabets Σ and Π. Since our work is based on an indexed alphabet, the choice of alphabet will impact the results only if the parameter sets significantly modify the parameterized longest common prefix (*pLCP*) array or perhaps $pLCP_{max}$ or $pLCP_{\mu}$ (see Definition 7).

The main focus of the implementation is on our core *pSA* constructions: Algorithms 5 and 6. Recall that Algorithms 5 and 6 sort the p-suffixes of the n-length text T by sorting *pAC* codes that represent m-length blocks of the n p-suffixes. Since our results are theoretical in nature, we first needed to address two key issues in the practical implementation: (1) validate the choice of the given integer m and (2) precisely handle the arithmetic computations of the transition functions (Definitions 23 and 24). Not addressing these issues may result in an invalid *pSA* construction.

The first key concern is how to validate the choice of m. That is, the m should not be too large so that $\texttt{tag}(\texttt{pAC}(T[i \ldots i + m - 1]))$ can be represented by the **long double** data type. Since a *pAC* code is a floating point value, the choice of m should produce codes that are clearly differentiable, where two lexicographically different m-blocks should have non-equal *pAC* codes with the same ordering. Prior to constructing the *pSA* in our implementation, we gather the p-string T and compute $prevT = \texttt{prev}(T)$. Next, the unique symbols within *prevT* are identified as the alphabet to encode. Let $\texttt{unique\_symbols}(prevT)$ identify the number of unique symbols in *prevT*. Recall that our *pAC* coding assumes that symbols are equally probable; these symbols in *prevT* occur with probability $p = \texttt{unique\_symbols}(prevT)^{-1}$. Now, we

Table 7
Max of LCP values within data sets.

Fibonacci		Bible		Ecoli	
i	$pLCP_{max} = LCP_{max}$	i	$pLCP_{max} = LCP_{max}$	i	$pLCP_{max} = LCP_{max}$
10 000	5819	10 000	64	800 000	1345
20 000	10 944	410 000	193	1 600 000	1345
30 000	17 709	810 000	487	2 400 000	1345
40 000	22 289	1 210 000	487	3 200 000	1346
50 000	28 655	2 410 000	487	4 000 000	1811
		3 210 000	487	4 638 690	2815
		3 445 275	487		

Table 8
Mean of LCP values within data sets.

Fibonacci			Bible			Ecoli		
i	$pLCP_{\mu}$	LCP_{μ}	i	$pLCP_{\mu}$	LCP_{μ}	i	$pLCP_{\mu}$	LCP_{μ}
10 000	2566.7	2566.7	10 000	7.7	7.2	800 000	15.8	13.8
20 000	5044.2	5044.2	410 000	11.9	11.1	1 600 000	15.1	13.1
30 000	7744.3	7744.3	810 000	14.6	13.8	2 400 000	15.7	13.7
40 000	10 130.6	10 130.6	1 210 000	13.8	13.0	3 200 000	16.0	14.1
50 000	12 766.8	12 766.8	2 410 000	13.5	12.8	4 000 000	17.4	15.4
			3 210 000	13.4	12.6	4 638 690	19.3	17.4
			3 445 275	13.3	12.5			

Table 9
Max of LCP values within random data sets.

i	$\mathcal{N}(16, 8)$		$\mathcal{N}(24, 12)$		$\mathcal{U}(1, 64)$		$\mathcal{U}(1, 16)$	
	$pLCP_{max}$	LCP_{max}	$pLCP_{max}$	LCP_{max}	$pLCP_{max}$	LCP_{max}	$pLCP_{max}$	LCP_{max}
125 000	23	6	29	6	34	6	17	7
250 000	23	7	29	6	35	6	18	8
375 000	24	7	29	6	38	6	19	9
500 000	24	7	29	6	38	6	19	9
625 000	24	9	29	6	38	6	19	9
750 000	24	9	29	6	38	6	19	9
875 000	26	9	29	6	38	6	19	9
1 000 000	26	9	29	6	38	6	19	9

Table 10
Mean of LCP values within random data sets.

i	$\mathcal{N}(16, 8)$		$\mathcal{N}(24, 12)$		$\mathcal{U}(1, 64)$		$\mathcal{U}(1, 16)$	
	$pLCP_{\mu}$	LCP_{μ}	$pLCP_{\mu}$	LCP_{μ}	$pLCP_{\mu}$	LCP_{μ}	$pLCP_{\mu}$	LCP_{μ}
125 000	13.2	2.9	15.2	2.5	17.9	2.2	11.1	3.6
250 000	13.6	3.1	15.6	2.7	18.4	2.3	11.5	3.8
375 000	13.9	3.2	15.9	2.8	18.7	2.5	11.7	4.0
500 000	14.1	3.2	16.1	2.9	18.9	2.6	11.8	4.1
625 000	14.2	3.3	16.3	2.9	19.1	2.6	12.0	4.2
750 000	14.3	3.4	16.4	3.0	19.2	2.7	12.1	4.2
875 000	14.4	3.4	16.5	3.0	19.3	2.7	12.1	4.3
1 000 000	14.5	3.5	16.6	3.0	19.4	2.8	12.2	4.3

generate the codes pAC_1 and pAC_2 for two lexicographically neighboring m-length p-suffixes and the difference is calculated as $\epsilon = |\texttt{tag}(pAC_1) - \texttt{tag}(pAC_2)| > 0$. We test to ensure that these codes are indeed separated by $\epsilon > 0$. Any codes that differ by less than ϵ are considered equal. An invalid ϵ (perhaps $\epsilon \leqslant 0$) causes different codes to collide and thus, signifies that a smaller m is required.

The second key concern is to handle the arithmetic computations of the transition functions (Definitions 23 and 24). In Algorithms 5 and 6, there is a separate section where the *pAC* floating point codes are initially generated with the `pAC` function (Algorithm 4) and subsequently created by the transition functions. In practice, successive floating point computations yield error and in our case, we rely heavily on the ordering of the *pAC* codes. Propagating these errors will eventually invalidate the ordering of the *pAC* codes and thus, result in an invalid *pSA*. Because we compare codes with an inequality based on ϵ, there is room for some error. We can handle this situation a few ways: renormalizing the floating

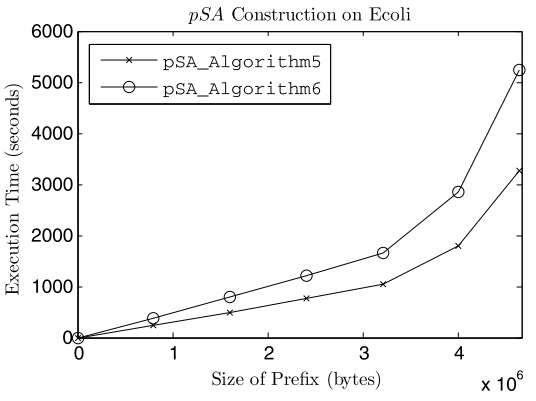

Fig. 2. Time for *pSA* construction on Ecoli sequence with $|\Sigma| = 0$, $|\Pi| = 4$, $m = 3$, and `unique_symbols(prev(T)) = 70`.

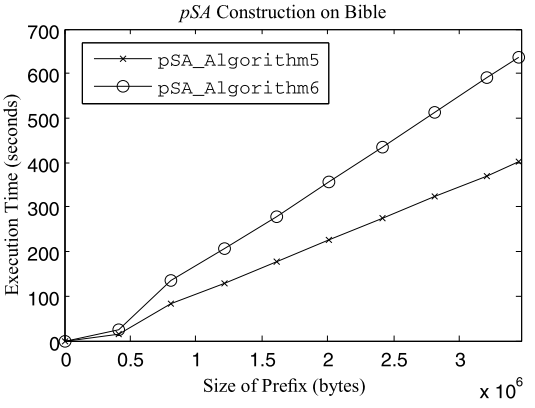

Fig. 3. Time for *pSA* construction on Bible with $|\Sigma| = 27$, $|\Pi| = 14$, $m = 3$, and `unique_symbols(prev(T)) = 3498`.

point codes or resetting the code cleanly with the function `pAC` after a specified number of transitions. For the purposes of this implementation, we have chosen the latter solution.

We first considered the sequence Ecoli from the Large Corpus. In our experiment, we construct the *pSA* for prefixes of the string. For this text, we considered the entire alphabet to be parameters: $\Sigma = \emptyset$ and $\Pi = \{a, c, g, t\}$. Table 7 shows that the standard LCP_{max} and the $pLCP_{max}$ are equal whereas Table 8 shows that $pLCP_{\mu} > LCP_{\mu}$ in all cases. We executed Algorithms 5 and 6 on the data. The results are displayed in Fig. 2. First, we notice that, in practice, the theoretically improved Algorithm 6 is not better than its predecessor Algorithm 5. Though Algorithm 6 is a theoretical improvement, the need to prepare the data for the **merge_sort** function is an implementation step that appears to be costly. In both cases, it is clear that the plots are linear until the final two points. By observing Table 7, we notice that the $pLCP_{max}$ is nearly constant as the size of i increases until the final two values of i, in which the value increases significantly. Since our *pSA* constructions sort *pAC* codes that represent p-suffix blocks until they differentiate, we should expect to sort more at the end of the text. So, (1) the constant $pLCP_{max}$ and the beginning linearity of the plot and (2) the ultimate complexity change in the plot and the simultaneous change in $pLCP_{max}$ are connected.

Next, we considered the Bible text from the Large Corpus. Since the Bible is composed of words, it is inappropriate to use individual symbols as parameters. So, the original Bible text was preprocessed and transformed into a text more suitable for p-matching and our program. First, only letters and spaces in the original text remained in the transformed text. All letters were forced to lowercase letters. Next, a unigram was constructed and each word appearing in the new text with a frequency $f \geqslant 7500$ was replaced with a unique uppercase symbol. (Thus, the size of the transformed Bible is slightly smaller than the original.) For example, the word `the` was replaced with the letter `L`. These 14 frequent words were considered parameters. Since the parameter words are now replaced with unique symbols not used by constants, there is no real need to adjust the remaining words since the remaining symbols must match anyway to detect a constant. So, we have constructed a new p-match problem where frequent words may be substituted. Tables 7 and 8 display information regarding the *pLCP* and *LCP* of the transformed Bible. We notice that slightly $pLCP_{\mu} > LCP_{\mu}$ in all cases and further, $pLCP_{max} = 487$ from $i = 810\,000$ until the end of the string. This is reflected in the results of Fig. 3 since the increase of each step is mirrored for each i where $pLCP_{max} = 487$. Even though both Algorithms 5 and 6 are linear in this case, the implementation of Algorithm 6 is again inferior, as was described in the Ecoli experiment. For concision, we further omit the Algorithm 6 results in future figures.

The qth Fibonacci sequence (or Fibostring) [28] is denoted by f_q and satisfies the recurrence $f_0 = b$, $f_1 = a$, and $f_q = f_{q-1} f_{q-2}$ for $q \geqslant 2$. The recurrence relation makes the resulting string naturally repetitive. Table 7 shows the LCP_{max} for the traditional case $\Sigma = \{a, b\}$ and $\Pi = \emptyset$ and $pLCP_{max}$ for the case $\Sigma = \emptyset$ and $\Pi = \{a, b\}$. The values LCP_{μ} and $pLCP_{\mu}$ are shown in Table 8. Notice that the alphabet set does not alter these values. Because our algorithms sort blocks until the *pAC* codes differ, these significant longest common prefix values signify that the Fibonacci string will force a worst case situation for our algorithms. We executed our Algorithm 5 implementation on prefixes of a Fibonacci string with $\Sigma = \emptyset$, $\Pi = \{a, b\}$, and varying m. The execution time is displayed in Fig. 4. Since this is considered a worst case string, we correctly anticipate from Theorem 25 that a fixed small m will result in a super-quadratic plot. The graph also confirms that as m increases, the execution time takes on a smaller complexity. We notice that the $pLCP_{max}$ occurs near the end of the tested Fibonacci sequence and at this point in the string, the graphs begin to clearly differentiate from one another. This clearly identifies the variation in complexity. Lastly, the memory requirements are displayed in Fig. 5. We see that memory requirements are linear regardless of the choice of m, as expected from Table 4 and a space analysis of Algorithm 5.

The next experiment tests our *pSA* construction on random sequences. Denote the Normal (Gaussian) distribution as $\mathcal{N}(a, b)$, where a is mean and b is variance. Denote the Uniform distribution as $\mathcal{U}(a, b)$ on the range $[a, b]$. These continuous distributions were discretized to yield discrete alphabet symbols. For a variety of distributions, a string of length $n = 1\,000\,000$ was generated. Algorithm 5 was executed on prefixes of the generated string with all parameters, $\Sigma = \emptyset$, and

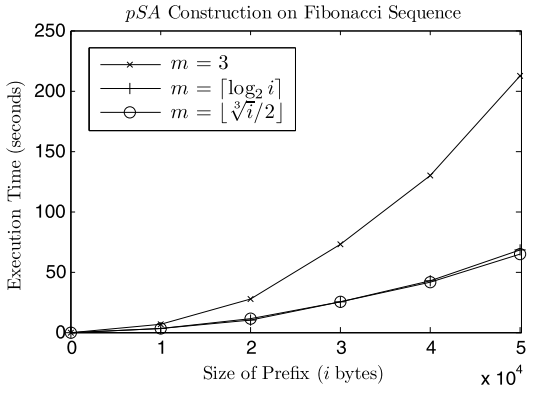

Fig. 4. Time for *pSA* construction on Fibonacci sequence with $|\Sigma| = 0$, $|\Pi| = 2$, and $\texttt{unique\_symbols}(\texttt{prev}(T)) = 5$.

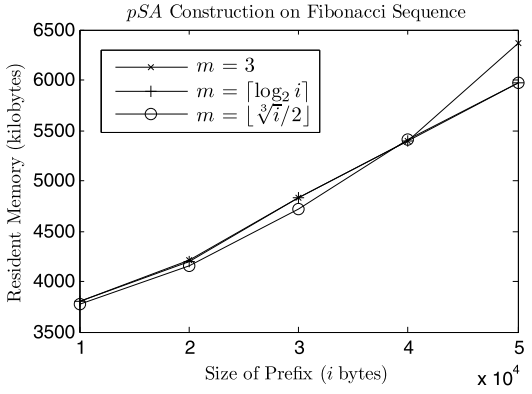

Fig. 5. Space for *pSA* construction on Fibonacci sequence with $|\Sigma| = 0$, $|\Pi| = 2$, and $\texttt{unique\_symbols}(\texttt{prev}(T)) = 5$.

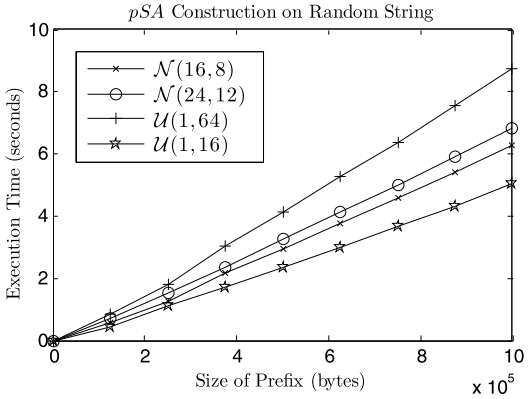

Fig. 6. Time for *pSA* construction on random sequences from Normal ($\mathcal{N}(a, b)$ where a is mean and b is variance) and Uniform ($\mathcal{U}(a, b)$ on the range $[a, b]$) discretized distributions with $|\Sigma| = 0$ and $m = 3$.

a fixed $m = 3$. Fig. 6 shows the results. The data in Tables 9 and 10 show that the introduction of parameters increases the values in every case. That is, $pLCP_{max} > LCP_{max}$ and $pLCP_{\mu} > LCP_{\mu}$ in every case. Since our algorithms successively sort blocks until *pAC* codes differ, this will increase the difficulty of the *pSA* construction when compared to standard *SA* construction. However, since the values $pLCP_{max}$ and $pLCP_{\mu}$ do not change significantly with increasing i, we expect that the *pSA* construction on the prefixes of the random strings will complete after a few sorts. This is confirmed in the results in Fig. 6, which shows a linear plot for the *pSA* construction of each random sequence.

In summary, our experimental results confirm that Algorithms 5 and 6 execute as expected from Theorems 25 and 26 respectively. Even though Algorithm 6 leads to the improved theoretical results in Theorem 26, the experimental results show that this implementation is inferior to Algorithm 5. We expect this to be a consequence of the extra implementation details required in Algorithm 6 to "setup" the array prior to passing the data to a standard **merge_sort** routine.

8. Conclusion and discussion

The notion that p-suffixes are dynamic limits our ability to *directly* sort them, i.e. obtaining the p-suffix array, using techniques for traditional suffix sorting such as successive doubling used in [24] or the conventional tricks used in induction [1,2,18,25]. Obtaining the p-suffix array *indirectly* from the p-suffix tree is subject to both the large memory footprint of the tree and the time/space tradeoff of the outgoing edge representation. These obstacles show the significance of the direct p-suffix sorting problem and present several intricacies and challenges to correctly and efficiently address the problem.

In terms of direct suffix sorting, the time/space tradeoff varies with algorithms. For traditional strings, approaches as those by [14] accomplish in-place direct suffix sorting in super-linear time, using only space for the suffix array and text. In other direct suffix sorting approaches, as this paper, we are concerned with improving the theoretical time complexity of the actual sorting and thus, we use a constant number of arrays in $O(n)$ space. We analyze our algorithms by incorporating multiple variables so that the algorithm may be tuned accordingly for practical use. We then analyze our algorithms experimentally in terms of specific characteristics of various data sets.

In this work, we use new and novel mechanisms to handle the intricacies of the direct p-suffix sorting problem by representing p-suffixes with fingerprints and arithmetic codes. First, we propose a theoretical algorithm using fingerprints

to p-suffix sort an n-length p-string in $O(n)$ time, with n and the alphabet size constrained in practice. Then, arithmetic codes are used to propose an algorithm to p-suffix sort p-strings in linear time on average. We further extend our average case result and propose a solution to sort p-suffixes in the worst case. This algorithm is improved by further observations. We display various cases in which our worst case algorithm performs in running times ranging from $O(n)$ to $O(n^2 \log n)$. Finally, we show that for a worst case p-string, it is possible to tune our algorithm to sort p-suffixes in $o(n^2)$ time. This addresses the open problem originally posed by [16] to prove the existence of sub-quadratic worst case p-suffix sorting solutions.

References

[1] D. Adjeroh, T. Bell, A. Mukherjee, The Burrows–Wheeler Transform: Data Compression, Suffix Arrays and Pattern Matching, Springer, New York, 2008.
[2] D. Adjeroh, F. Nan, Suffix sorting via Shannon–Fano–Elias codes, Algorithms 3 (2) (2010) 145–167.
[3] A. Amir, M. Farach, S. Muthukrishnan, Alphabet dependence in parameterized matching, Inf. Process. Lett. 49 (1994) 111–115.
[4] B. Baker, A theory of parameterized pattern matching: Algorithms and applications, in: STOC'93, ACM, New York, 1993, pp. 71–80.
[5] B. Baker, Parameterized pattern matching: Algorithms and applications, J. Comput. Syst. Sci. 52 (1) (1996) 28–42.
[6] B. Baker, Parameterized pattern matching by Boyer–Moore-type algorithms, in: SODA'95, ACM, Philadelphia, PA, 1995, pp. 541–550.
[7] R. Beal, Parameterized Strings: Algorithms and Data Structures, MS Thesis, West Virginia University, 2011.
[8] R. Beal, D. Adjeroh, Parameterized longest previous factor, Theor. Comput. Sci. 437 (2012) 21–34.
[9] R. Beal, D. Adjeroh, p-Suffix sorting as arithmetic coding, in: IWOCA'11, Springer, Heidelberg, 2011, pp. 44–56.
[10] R. Cole, R. Hariharan, Faster suffix tree construction with missing suffix links, in: STOC'00, ACM, New York, 2000, pp. 407–415.
[11] T. Cover, J. Thomas, Elements of Information Theory, Wiley, 1991.
[12] S. Deguchi, F. Higashijima, H. Bannai, S. Inenaga, M. Takeda, Parameterized suffix arrays for binary strings, in: PSC'08, Czech Republic, 2008, pp. 84–94.
[13] L. Devroye, W. Szpankowski, B. Rais, A note on the height of suffix trees, SIAM J. Comput. 21 (1992) 48–53.
[14] G. Franceschini, S. Muthukrishnan, In-place suffix sorting, in: ICALP'07, Poland, 2007, pp. 533–545.
[15] D. Gusfield, Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology, Cambridge University Press, Cambridge, UK, 1997.
[16] T. I, S. Deguchi, H. Bannai, S. Inenaga, M. Takeda, Lightweight parameterized suffix array construction, in: IWOCA'09, in: LNCS, vol. 5874, Springer, Heidelberg, 2009, pp. 312–323.
[17] R. Idury, A. Schäffer, Multiple matching of parameterized patterns, Theor. Comput. Sci. 154 (1996) 203–224.
[18] J. Kärkkäinen, P. Sanders, S. Burkhardt, Linear work suffix array construction, J. ACM 53 (2006) 918–936.
[19] S. Karlin, G. Ghandour, et al., New approaches for computer analysis of nucleic acid sequences, PNAS 80 (18) (1983) 5660–5664.
[20] R. Karp, M. Rabin, Efficient randomized pattern-matching algorithms, IBM J. Res. Dev. 31 (1987) 249–260.
[21] S. Kosaraju, Faster algorithms for the construction of parameterized suffix trees, in: FOCS'95, ACM, Washington, DC, 1995, pp. 631–637.
[22] T. Lee, J. Na, K. Park, On-line construction of parameterized suffix trees, in: SPIRE'09, Springer, Heidelberg, 2009, pp. 31–38.
[23] T. Lee, J. Na, K. Park, On-line construction of parameterized suffix trees for large alphabets, Inf. Process. Lett. 111 (5) (2011) 201–207.
[24] U. Manber, G. Myers, Suffix arrays: A new method for on-line string searches, SIAM J. Comput. 22 (1993) 935–948.
[25] G. Manzini, P. Ferragina, Engineering a lightweight suffix array construction algorithm, Algorithmica 40 (2004) 33–50.
[26] A. Moffat, R. Neal, I. Witten, Arithmetic coding revisited, ACM Trans. Inf. Syst. 16 (1995) 256–294.
[27] T. Shibuya, Generalization of a suffix tree for RNA structural pattern matching, Algorithmica 39 (1) (2004) 1–19.
[28] W. Smyth, Computing Patterns in Strings, Pearson, New York, 2003.
[29] B. Zeidman, Software v. software, IEEE Spectr. 47 (2010) 32–53.